Profile-Guided Temporal Prefetching

Mengming Li
HKUST
mengming.li@connect.ust.hk

Qijun Zhang
HKUST
qzhangcs@connect.ust.hk

Yichuan Gao
Intel
yichuan.gao@intel.com

Wenji Fang
HKUST
wfang838@connect.ust.hk

Yao Lu
HKUST
yludf@connect.ust.hk

Yongqing Ren
Intel
yongqing.ren@intel.com

Zhiyao Xie*
HKUST
eezhiyao@ust.hk

Abstract

Temporal prefetching shows promise for handling irregular memory access patterns, which are common in data-dependent and pointer-based data structures. Recent studies introduced on-chip metadata storage to reduce the memory traffic caused by accessing metadata from off-chip DRAM. However, existing prefetching schemes struggle to efficiently utilize the limited on-chip storage. An alternative solution, software indirect access prefetching, remains ineffective for optimizing temporal prefetching.

In this work, we propose Prophet—a hardware-software co-designed framework that leverages profile-guided methods to optimize metadata storage management. Prophet profiles programs using counters instead of traces, injects hints into programs to guide metadata storage management, and dynamically tunes these hints to enable the optimized binary to adapt to different program inputs. Prophet is designed to coexist with existing hardware temporal prefetchers, delivering efficient, high-performance solutions for frequently executed workloads while preserving the original runtime scheme for less frequently executed workloads. Prophet outperforms the state-of-the-art temporal prefetcher, Triangel, by 14.23%, effectively addressing complex temporal patterns where prior profile-guided solutions fall short (only achieving 0.1% performance gain). Prophet delivers superior performance across all evaluated workload inputs, introducing negligible profiling, analysis, and instruction overhead.

CCS Concepts

• **Computer systems organization → Architectures**.

Keywords

Temporal Prefetching, Profile-Guided Optimization

*Corresponding Author

ACM Reference Format:
Mengming Li, Qijun Zhang, Yichuan Gao, Wenji Fang, Yao Lu, Yongqing Ren, and Zhiyao Xie. 2025. Profile-Guided Temporal Prefetching. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731070

1 Introduction

Data prefetching, a widely studied technique for addressing the "memory wall" [59], has been extensively researched to enhance processor performance. Among various data prefetching techniques [7, 9–13, 20, 23–26, 30, 31, 37–39, 41, 42, 45, 46, 49, 50, 52, 53, 55–58], temporal prefetching [7, 10, 26, 46, 55, 57, 58] shows particular promise for addressing irregular memory access patterns, which often arise from indirect/data-dependent memory accesses and pointer-based data structures. Temporal prefetchers typically require significant metadata storage to record correlations between memory addresses, making efficient metadata storage design vitally important. Recent studies [7, 56, 57] propose relocating metadata storage from off-chip DRAM [10, 26, 46, 55, 58] to the on-chip metadata table within LLC to reduce memory traffic. Since on-chip storage is a limited resource, efficient management of the metadata table becomes even more critical.

Hardware temporal prefetcher. Existing hardware temporal prefetchers [7, 56, 57] employ various techniques for metadata table management, such as training data filtering, replacement policies, and resizing. However, they fail to balance performance gains with storage overhead. For example, Triage [57] adopts an advanced replacement policy (e.g., Hawkeye [27]) that incurs a 13 KB storage overhead but results in only a 0.25% performance gain. The state-of-the-art temporal prefetcher, Triangel [7], introduces additional techniques like training data filtering to improve metadata table management. However, according to its own ablation study [7], Triangel's performance gain mostly comes from aggressive prefetching instead of its metadata table management, which actually incurs 90% of the storage overhead.

Ideal metadata table management involves two key requirements: (1) storing all metadata that contributes to useful prefetches in the metadata table; and (2) filtering out metadata that does not contribute to useful prefetches. However, existing temporal prefetchers [7, 56, 57] fail to balance these two requirements. They either store many invalid metadata entries (e.g., no insertion policy for Triage

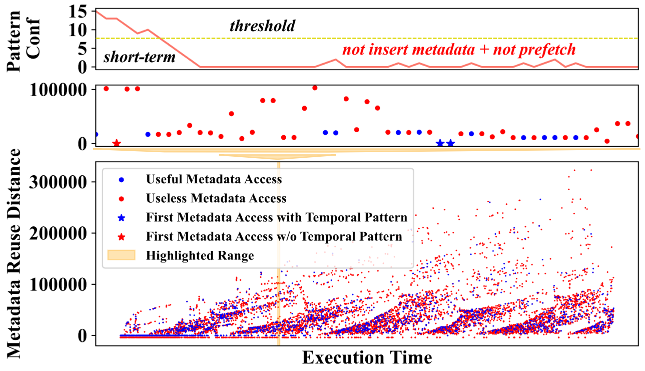


Figure 1: The bottom figure shows a metadata access pattern: 1) Blue/Red *dots* are metadata accesses that result in useful/useless prefetches; 2) Blue/Red *stars* represent first metadata access with/without temporal patterns. Their corresponding metadata should/should not be inserted in the metadata table. The top figure shows how Triangel [7] applies its *PatternConf* to the highlighted metadata access pattern.

[57]) or incorrectly filter out metadata entries that could result in useful prefetches (e.g., overly conservative insertion policy for Triangel [7]).

To investigate why Triangel's metadata management is inefficient, Figure 1 shows a metadata access pattern[1,2] and analyzes how Triangel applies its *PatterConf* to the pattern. The 4-bit *PatternConf* evaluates whether future memory accesses exhibit temporal patterns. Triangel utilizes past short-term data to update the *PatternConf*. Specifically, useful metadata accesses (blue dots) increase the *PatternConf*, while useless metadata accesses (red dots) decrease it. When *PatternConf* falls below a threshold, indicating the absence of temporal patterns in the future, Triangel will completely disable metadata insertion (no insertion for stars in Figure 1).

We have two observations for Figure 1: (1) The metadata access patterns of temporal prefetching are highly variable, characterized by interleaved useful (blue) and useless (red) metadata accesses and large metadata reuse distance variance. (2) Triangel does not adapt to such dynamic variance of temporal patterns. As shown in Figure 1, *PatterConf* drops to 0 due to the occurrence of many red dots in a short term. As a result, Triangel incorrectly rejects the insertion of subsequent interleaved blue stars (first metadata access with temporal pattern). Similar challenges also affect other metadata management strategies, as we will cover in Section 2.1.

Purely hardware-based temporal prefetching fails to handle the dynamic variance of metadata accesses due to two main reasons: (1) they lack visibility into future program behavior, and (2) while long-term execution data could help mitigate this problem, storing and analyzing such data would introduce significant performance and storage overhead.

Profile-guided temporal prefetcher. To address the above challenges, profile-guided techniques are promising by leveraging comprehensive program execution data to guide metadata table management strategies. However, existing profile-guided solutions

[1]This pattern is derived from a hardware temporal prefetcher with an unlimited metadata table size and no insertion policy.
[2]This pattern is extracted from a frequently accessed instruction in omnetpp, a workload where Triangel shows limited effectiveness.

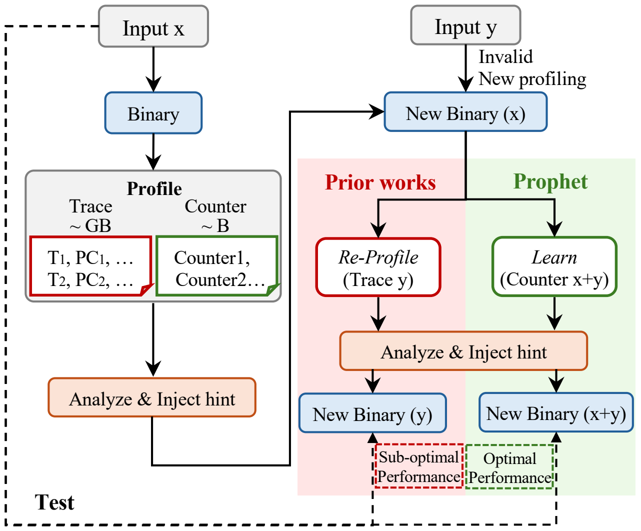


Figure 2: Comparison between Prophet and prior profile-guided solutions. Prophet is *lightweight* as it only uses counters for profiling. Prophet can integrate counters from multiple inputs, enabling it to *adapt to* varying program inputs.

[6, 29, 33, 60] for prefetching are *ineffective for most temporal patterns*. These solutions primarily focus on inserting software prefetch instructions for indirect memory accesses *where the prefetch kernel follows a regular stride pattern*. They struggle to handle more complex irregular patterns, such as pointer-chasing accesses and indirect memory accesses with complicated prefetch kernels. This limitation arises primarily because many irregular patterns involve long-chain dependencies [8], and computing dependent addresses along the chain significantly impacts prefetching timeliness.

Our solution named Prophet. Driven by our analysis, we propose `Prophet`[3]—a hardware-software co-designed framework, which maintains the metadata table within hardware temporal prefetchers while leveraging profile-guided methods to improve metadata table management. Prophet addresses aforementioned challenges without requiring significant hardware overheads: it offloads hardware-intensive tasks (e.g., data analysis) to software and injects hints into programs to guide the metadata table *insertion policy*, *replacement policy*, and *resizing operations*.

Beyond the scope of temporal prefetching, Prophet introduces an innovative, efficient profile-guided solution that overcomes key challenges in traditional profile-guided methods, as outlined below.

- **Adaptable.** We investigate why profile-guided optimizations often struggle with varying program inputs and why hints derived from one input may not apply to others. Based on these insights, we enable Prophet to integrate profiling data from multiple program inputs. Leveraging the aggregated data, Prophet can generate a single optimized binary that adapts effectively across these inputs, as shown in Figure 2.
- **Lightweight.** Prophet leverages Performance Monitoring Unit (PMU) counters instead of traces to profile programs, offering two major benefits: (1) it avoids the significant performance and storage overhead linked to trace-based profiling;

[3]Prophet is open-sourced at: https://github.com/hkust-zhiyao/Prophet.

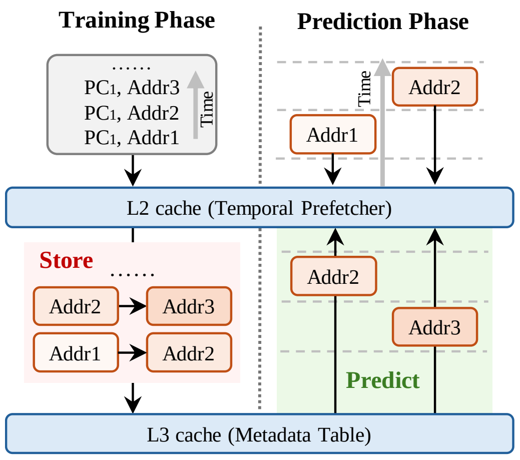


Figure 3: A general framework of temporal prefetching.

(2) it is readily applicable to current architectures without requiring additional memory trace systems.

- **Compatible.** Prophet can co-exist with existing hardware temporal prefetchers. Prophet offers high-performance yet efficient solutions for frequently executed workloads while maintaining the original runtime solution (e.g., Triangel [7]) for rarely executed workloads.

We evaluate Prophet on applications representative of temporal patterns, which are commonly used in prior studies [7, 56–58]. Across all applications, Prophet outperforms the state-of-the-art software indirect memory access prefetching schemes, RPG^2 [60] by 34.48% and hardware temporal prefetcher, Triangel [7] by 14.23%. This performance gain is driven by Prophet's efficient metadata table management, which not only significantly enhances prefetching coverage but also maintains high prefetching accuracy. Extensive evaluations demonstrate that Prophet can adapt to varying program inputs while introducing negligible profiling, analysis, and instruction overheads.

2 Background

2.1 Hardware Temporal Prefetching

As shown in Figure 3, the core idea of hardware temporal prefetching [7, 10, 26, 46, 55–58] is to record previously accessed memory addresses and their correlations, referred to as metadata. When recorded addresses are re-accessed, the prefetcher can use the correlation information to predict future memory accesses. To record metadata of various temporal patterns, hardware temporal prefetching algorithms require significant data storage. Early temporal prefetchers [10, 26, 46, 55, 58] utilized DRAM to store the metadata. However, fetching metadata from DRAM consumes a substantial amount of memory bandwidth that could otherwise be used for demand memory accesses. To address this issue, Triage [56, 57] recently proposed storing metadata in a Markov table that shares space with LLC, eliminating the need to load metadata from off-chip memory. However, on-chip storage is precious, so temporal prefetchers must operate with a limited metadata table size.

To efficiently utilize on-chip memory, Triage [56, 57] and the state-of-the-art temporal prefetcher, Triangel [7], introduce several techniques for managing the metadata table:

2.1.1 Insertion policy (training data filtering). Not every demand request should be used to train the temporal prefetcher and trigger metadata insertion. To improve the utilization of the metadata table, temporal prefetchers should focus on handling demand requests that exhibit *solvable* temporal patterns. The "solvable" indicates these patterns repeat in a short enough sequence to fit in the metadata table. Ideally, the temporal prefetcher should reject metadata insertion for demand requests that do not meet this criterion.

The first on-chip temporal prefetching scheme, Triage, does not implement any insertion (i.e. filtering) policy. The state-of-the-art temporal prefetcher, Triangel, leverages *PatternConf* and *ReuseConf* to identify and filter out demand requests that do not fall within the capabilities of temporal prefetchers. The *PatternConf* checks whether demand requests from a memory instruction exhibit a temporal pattern. The *ReuseConf* further evaluates whether the temporal pattern can fit the metadata table.

Inefficiency of insertion policy. According to Triangel's ablation study [7], filtering out demand requests with *PatternConf* and *ReuseConf* only yields marginal overall performance improvements and even degrades the performance for many applications. This limitation stems from the inaccuracy of these metrics (i.e., *PatternConf* and *ReuseConf*), as illustrated in Figure 1. As a result, inaccurate filtering discards metadata that could lead to useful prefetches.

Insertion policy in Prophet. Prophet implements a more accurate insertion policy, filtering out only metadata that is highly unlikely to originate from temporal patterns. This is accomplished by identifying PCs whose overall prefetching accuracy falls below an extremely low threshold, a metric easily obtained during the profiling stage. Unlike hardware temporal prefetchers, which rely on short-term data to guide future program execution, the metrics gathered during profiling (e.g., prefetching accuracy) reflect the actual behavior of programs, resulting in more precise decisions.

2.1.2 Replacement Policy. When the metadata table capacity is insufficient to accommodate a new entry, the temporal prefetcher must decide which existing metadata entry to evict. Triage [57] observed that only a small fraction of metadata is frequently reused. As a result, it employs an advanced replacement policy (e.g., Hawkeye [27]) to evict metadata entries that are less likely to be accessed in the future, thereby enhancing the utilization of metadata tables. However, according to Triangel [7], this replacement policy provides only marginal performance gains, achieving a speedup of less than 0.25%. As a result, Triangel replaces Hawkeye in Triage with a simpler replacement policy—SRRIP [28]—to balance storage overhead with performance gains.

Inefficiency of replacement policy. (1) The reuse distance of metadata entries varies significantly, as shown in Figure 1. These replacement policies solely focus on predicting reuse distances, making them inefficient for handling such high variance [54]. (2) They focus solely on increasing hits in metadata table without considering if metadata hits further result in useful prefetches.

Replacement policy in Prophet. Prophet enhances the replacement policy by incorporating prefetching accuracy as an additional metric for selecting victim entries in the metadata table. After filtering out metadata through the insertion policy, Prophet assigns different priority levels to the remaining metadata based on the prefetching accuracy. Metadata entries that are less likely to generate useful prefetch requests are assigned lower priority levels, making them prioritized candidates for replacement. Like the insertion policy, Prophet obtains prefetching accuracy during the profiling stage, ensuring precise and effective management.

2.1.3 Resizing. The metadata table shares space with the LLC, requiring a careful balance in space allocation to avoid compromising the LLC's ability to serve regular demand requests. Previous schemes address this trade-off using methods like Bloom Filters [16] or Set Dueller [7]. Triage employs a Bloom Filter to calculate the effective entries in the metadata table, but tracking approximately 200,000 entries incurs a storage overhead exceeding 200 KB. To mitigate this, Triangel introduces the Set Dueller, which uses a small subset of cache sets to model the full-size regular LLC and Markov table. The Set Dueller works by simulating various partitioning configurations for the cache and the Markov table, evaluating their respective hit rates. After a defined window, the Set Dueller selects the configuration that maximizes the hit rate. This approach reduces resizing overhead to approximately 2 KB by tracking only selected cache sets instead of the entire metadata table.

Inefficiency of resizing. Similar to insertion and replacement policies, the hit rate between LLC and metadata table sampled by Triangel's resizing sometimes fails to accurately predict future metadata table requirements. For example, in workloads like omnetpp and mcf, we observe that Triangel's resizing often chooses overly conservative metadata table sizes. Although this improves prefetching accuracy, it reduces prefetching coverage, ultimately damaging overall system performance. As a result, Set Dueller provides only limited performance benefits for Triangel.

Resizing in Prophet. Prophet leverages profile-guided techniques to implement a Bloom-Filters-like method, which helps maintain precision while avoiding significant storage overhead introduced by runtime approaches. Specifically, Prophet allocates storage to the metadata table based on the peak metadata usage observed during the profiling stage. This approach is adopted because resizing provides only marginal performance gains (Section 5.9), while incorrect resizing can significantly degrade performance.

2.2 Software Indirect Access Prefetching

Software indirect access prefetching [6, 17, 19, 22, 29, 33, 60] predicts indirect memory accesses by inserting prefetch instructions into programs. Indirect memory accesses include prefetch kernel accesses (e.g., *b[i])* and data-dependent memory accesses linked to that prefetch kernel (e.g., *a[b[i]]*).

Software indirect access prefetching primarily targets situations where the prefetch kernel follows a stride pattern, such as when the accesses for a[b[i]] occur in loops with indices *i, i + d, i + 2d*, and so on. These schemes generally follow a three-step process: (1) identifying the prefetch kernel; (2) calculating the prefetch distance between the insertion position of software prefetch instructions and the prefetch kernel; and (3) inserting the software prefetch instructions into original programs. The most critical aspect is calculating the prefetching distance, as the position of prefetch instructions determines the prefetching timeliness.

Based on the methods used to calculate the prefetching distance, previous works can be categorized into static solutions [17, 19, 22] and profile-guided optimization (PGO) solutions [6, 29, 33, 60]. In static solutions, programmers manually determine where to insert software prefetch instructions. However, due to the complexity of indirect memory accesses, manual insertion is prone to errors and may cause performance slowdowns. Profile-guided solutions overcome the limitations of static approaches. They profile running programs to automatically identify optimal positions for inserting prefetch instructions. Experimental results [29, 60] indicate that these approaches can significantly improve system performance on certain graph benchmarks, such as CRONO [5].

Inefficiency of software indirect access prefetching. Existing software indirect access prefetching schemes are effective only for a narrow subset of indirect memory accesses where the prefetch kernel follows a regular stride pattern. They fail to address most irregular patterns effectively, including complex indirect accesses (e.g., prefetch kernel without stride patterns) and pointer-chasing accesses. This limitation arises primarily because many irregular patterns involve long-chain dependencies [8], and computing dependent addresses along the chain significantly affects the prefetching timeliness.

To validate their limitations, in Section 5.2, we evaluate the state-of-the-art PGO-based scheme, RPG^2 [60], on representative SPEC CPU workloads commonly used in temporal prefetching studies [7, 56, 57]. Normalized to a baseline without a temporal prefetcher, RPG^2 achieves only a 0.1% performance improvement, significantly lower than its gains on graph benchmarks. We observe that this underperformance is due to the complexity of indirect memory accesses in the evaluated workloads. For instance, in mcf, the index of a prefetch kernel is derived through a series of logical operations and multi-step arithmetic computations.

Solution in Prophet. Prophet is applicable to all types of temporal patterns because it gets rid of software prefetch instruction. Prophet only guides the execution of hardware structures with hints. By preserving the core functions of the metadata table, Prophet maintains the ability to handle all temporal patterns like hardware temporal prefetchers.

3 Overview

3.1 Architecture Overview

This section provides an overview of Prophet architecture, which is **compatible** with existing hardware temporal prefetchers. For the metadata format, Prophet packs 12 compressed metadata entries inside each 64-byte cache line, with each metadata entry containing a 10-bit tag and a 31-bit target address. For the metadata table management, Prophet offers efficient, high-performance solutions for frequently executed workloads while maintaining the original runtime solution (e.g., Triangel) for rarely executed workloads. As shown in Figure 4, Prophet consists of three components: profile-guided insertion policy, profile-guided replacement policy (along with its associated replacement states), and profile-guided resizing operations. These profile-guided components rely on two types of information granularity: application-level and PC-level. The application-level information is embedded in the Control and Status Register (CSR). The PC-level information is accompanied by demand requests, referred to as *Hint* in Figure 4. Next, we will overview these components:

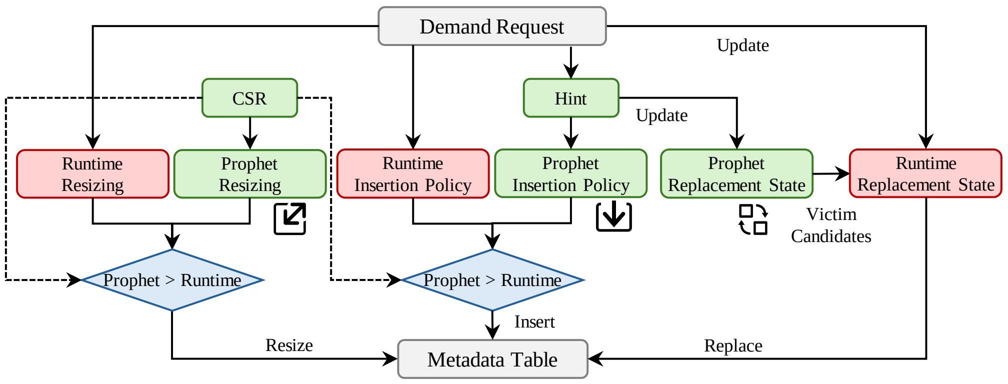


Figure 4: Prophet architecture overview. Prophet coexists with hardware temporal prefetchers by sharing the same metadata table but leveraging more accurate profile-guided methods for metadata table management.

Prophet Insertion Policy: This policy determines whether to train the temporal prefetcher with demand requests and insert their associated metadata into the table by checking the hint information carried by the demand requests. We disable the Runtime Insertion Policy when the Prophet Insertion Policy is enabled. Prophet's building blocks are activated through CSR manipulation. Specifically, after profiling the program, a CSR manipulation instruction is inserted at the beginning of the binary to enable Prophet.

Prophet Replacement Policy: This policy assigns replacement priorities to demand requests, with the priority information carried by hints within demand requests. Upon inserting new metadata, we record the priority information from hints into the Prophet Replacement State. During the replacement process, the Prophet Replacement Policy first generates candidate victims for the Runtime Replacement Policy, which then chooses the final victim.

Prophet Resizing: This policy retrieves the target metadata table size from the CSR and allocates LLC space to the metadata table at the beginning of program execution. Similar to the Prophet Insertion Policy, we disable the Runtime Resizing when Prophet Resizing is enabled.

Compatibility. In our framework, Prophet coexists with hardware temporal prefetchers by sharing the same metadata table but leveraging more accurate profile-guided methods for metadata table management. Additionally, Prophet reuses the runtime solution's replacement states, integrating both reuse distance and prefetching accuracy into its replacement policy for enhanced effectiveness. Programmers can switch between Prophet and the hardware temporal prefetcher based on application execution frequency and the trade-off between Prophet's performance gains and its impact on DRAM traffic (Section 5.9).

3.2 Process Overview

Figure 5 outlines the process flow of Prophet, comprising three steps: Profiling, Analysis, and Learning. Next, we overview the operations involved in each step of Prophet's process:

Step1: Profiling (Section 4.1). Prophet executes target binaries with the simplified temporal prefetcher to collect counters through the user-space PMU interfaces, such as Linux's perf tools [21]. These collected counters are then analyzed in the subsequent *Analysis* step to derive optimized metadata table management strategies. The *simplified temporal prefetcher* operates with a configuration of Prophet with insertion policy disabled, a fixed metadata table

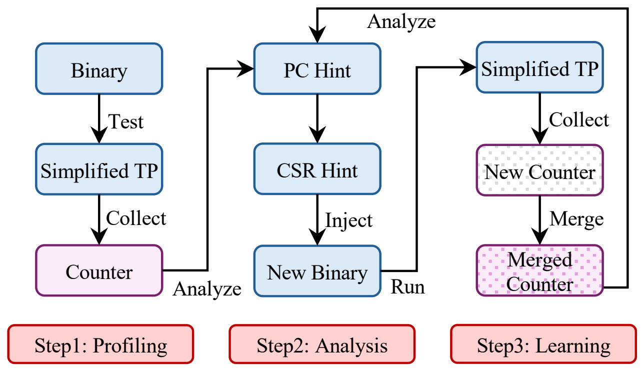


Figure 5: Prophet process overview. Step1: Prophet leverages the PMU to gather counters related to the temporal prefetcher's performance. Step2: Prophet analyzes the collected counters to generate hints and then injects hints into the original binaries. Step 3: Prophet samples and learns counters across different program inputs.

of 1 MB, and a prefetching degree of 1. This configuration ensures an unbiased evaluation of memory instructions under temporal prefetching, without incorporating any additional optimizations. Compared to other profile-guided solutions [29, 32, 35, 36, 54] that uses trace for profiling, Prophet is more ***lightweight***, introducing negligible profiling overhead in the Profiling step, as well as analysis and instruction overhead in the Analysis step (Section 5.4).

Step2: Analysis (Section 4.2). Prophet processes the counters collected in Step 1 through offline scripts, generating two types of hints for efficient metadata table management: PC-level and application-level. PC-level hints are specific to individual memory instructions and are utilized in Prophet's insertion policy and replacement policy. On the other hand, application-level hints are applied globally through a CSR manipulation instruction at the program's start and are used in Prophet's resizing operations. These hints are injected into the original binary, resulting in an optimized binary that can execute with Prophet.

Step3: Learning (Section 4.3). At regular intervals, Prophet samples new counters from varying inputs with the simplified Prophet and integrates them with previously collected counters. Then, the subsequent Analysis step will generate new hints based on the merged counters, making Prophet's insertion policy, replacement policy, and resizing operations ***adaptable*** to all encountered

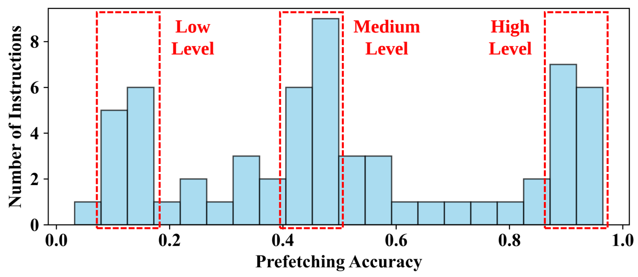


Figure 6: The prefetching accuracy of temporal prefetching across different memory instructions in omnetpp.

program inputs. Through repeated learning, Prophet innovatively enables a single optimized binary to achieve optimal performance across a wide range of program inputs.

4 Design

4.1 Step 1: Profiling

Two key questions arise in this step: *What information is necessary to efficiently guide the management of the metadata table? How can we acquire this information with current architectures?* To answer the first question, we identify two primary objectives for managing the metadata table: (1) enhancing its utilization, and (2) reducing its impact on the LLC. The *insertion* and *replacement* policies are well-suited for achieving the first goal, while *resizing* operations are suitable for the second. For enhancing utilization, **prefetching accuracy per memory instruction** serves as a critical metric. As shown in Figure 6, although individual metadata accesses (Figure 1) exhibit high variability, the temporal prefetching accuracy of every instruction can be broadly classified into distinct levels. Since the prefetching accuracy reflects the adaptability of memory instructions to the temporal prefetcher, lower-level memory instructions generate fewer memory accesses exhibiting temporal patterns compared to higher-level memory instructions. Thus, the insertion policy can filter out metadata entries from lowest-level memory instructions, while the replacement policy can assign fine-grained replacement priorities to unfiltered metadata entries based on their levels. For reducing the metadata table's impact on the LLC, the **number of allocated entries** in the metadata table is a useful metric. Sampling this metric at the program execution's end allows us to determine the maximum metadata table size.

We employ Intel's Processor Event-Based Sampling (PEBS) [3] utility to collect PC-level prefetching accuracy. PEBS records the program context (e.g., PC) when specific events occur. Prior to data collection, we configure the temporal prefetcher in its simplified mode (Section 3.2) and disable all other L2 prefetchers. We then enable PEBS to sample following events:

- `MEM_LOAD_RETIRED.L2_Prefetch_Issue` counts the number of issued prefetch requests.
- `MEM_LOAD_RETIRED.L2_Prefetch_Useful` tracks the number of prefetches hit by demand requests.

The above two events can be implemented with minor modifications to existing `MEM_LOAD_RETIRED.L2_MISS` (L2 cache miss) event, which is already supported on Intel's Xeon Processor [1]. Each PC's prefetching accuracy can be computed as:

$$Prefetching\ Accuracy = \frac{L2\_Useful\_Prefetches}{L2\_Issued\_Prefetches}$$

The number of allocated entries in the metadata table, an application-level metric, can be measured using standard PMU counters. For instance, we can define two counters: one for the number of metadata table insertions and another for replacements. The number of allocated entries in the metadata table can be calculated as:

$$Allocated\ Entries = Insertions - replacements$$

4.2 Step 2: Analysis

This step analyzes the counters collected in Step 1 to generate PC-level and application-level hints for the insertion policy, replacement policy, and resizing operations. Although both the insertion and replacement policies are guided by prefetching accuracy, Prophet adopts different strategies. The insertion policy filters out memory instructions that clearly lack temporal patterns (those with extremely low prefetching accuracy), while the replacement policy applies more refined management to the remaining memory instructions. Our observations indicate that as long as a memory instruction's prefetching accuracy is not particularly low, at least some of its memory accesses exhibit a temporal pattern.

Prophet Insertion Policy uses Equation 1 to decide whether to use demand requests from PCs to train the temporal prefetcher and insert the corresponding metadata. We define *EL_ACC* as an extremely low threshold for the prefetching accuracy, showing that memory instructions almost exhibit no temporal pattern.

$$I(acc) = \begin{cases} 1, & acc \geq EL\_ACC \\ 0, & acc < EL\_ACC \end{cases} \tag{1}$$

The above equation indicates that if a PC's accuracy under temporal prefetching falls below *EL_ACC*, Prophet instructs the temporal prefetcher to discard all demand requests associated with that PC. This decision is encoded as a one-bit hint injected into the corresponding memory instructions. All demand requests generated by these instructions will carry the embedded hint. Upon reaching the prefetcher, a simple logic checks the hint to decide whether to discard the corresponding requests.

Prophet Replacement Policy assigns a priority level to each stored metadata entry, with lower levels prioritized for replacement. Like the Prophet insertion policy, these priority levels are initially embedded in memory access instructions. When inserting new metadata entries, Prophet records their associated priority levels into the Prophet Replacement State.

We apply Equation 2 to determine each memory instruction's priority level. The n is a parameter controlled by the designer.

$$R(acc) = \begin{cases} 0, & EL\_ACC \leq acc < \frac{1}{2^n} \\ 1, & \frac{1}{2^n} \leq acc < \frac{2}{2^n} \\ 2, & \frac{2}{2^n} \leq acc < \frac{3}{2^n} \\ \vdots & \vdots \\ 2^n - 1, & \frac{2^n-1}{2^n} \leq acc < 1 \end{cases} \tag{2}$$

When choosing victim entries in the metadata table, Prophet first identifies victim candidates with the lowest priority level. Then,

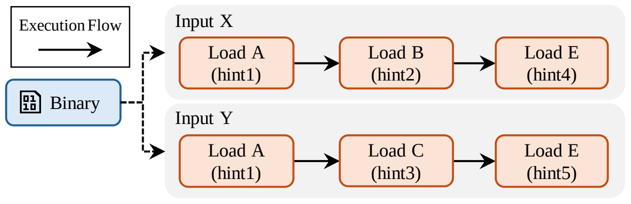


Figure 7: Challenges of traditional profile-guided optimizations across different program inputs.

Prophet applies LRU among these victim candidates to determine the final replaced entry.

Prophet Resizing estimates the metadata table size based on the number of allocated metadata entries at the end of program execution. Assuming the counter value is S, we first round it[4] to the nearest power of 2, and then use the Equation 3 to determine the number of ways allocated for the metadata table in the LLC:

$$Allocated\ Ways = ceil\left(\frac{Target\ Size}{Number\ of\ sets\ in\ LLC}\right) \tag{3}$$

We completely disable temporal prefetching when the outcome of the above equation is less than 0.5. Based on the result of Equation 3, Prophet inserts a CSR manipulation instruction at the program's start to configure the metadata table size.

4.3 Step 3: Learning

In this step, Prophet collects counters from varying program inputs and integrates them, allowing the optimized binary to deliver optimal performance across diverse program inputs. Figure 7 illustrates *why profile-guided methods are sensitive to different program inputs* and *why the hints derived from one input may not be applicable to other inputs.* A single binary may follow different execution paths depending on the input. For example, with input X, the binary might execute memory access instructions A, B, E, while input Y causes it to execute instructions A, C, E. When applying profile-guided optimizations to this binary under inputs X and Y respectively, there are three different scenarios:

- **Load A**: Under both inputs X and Y, the binary executes Load A. Profile-guided optimizations generate identical hints for both inputs, possibly because they execute the same code and produce similar profiling metrics. In this case, the hints generated for Load A under input X remain effective for Load A under input Y.
- **Loads B and C**: execute completely different instructions, resulting in distinct hint information. In this case, any hint derived from Load B under input X is ineffective for Load C under input Y.
- **Load E**: Although both inputs X and Y execute Load E, the global execution context impacts Load E differently under each input. Consequently, profile-guided methods may generate distinct hints. In this case, the hints generated for Load E under input X are ineffective for Load E under input Y.

Building on the three cases above, we develop a process that enables our counter-based profile-guided optimizations to adapt to different program inputs. Prophet maintains the counters from step 2 when it progresses to step 3 (e.g., input X). In step 3, we assume Prophet acquires new counters under previously unseen inputs (e.g., input Y). Prophet merges them with the previously maintained counters. For prefetching accuracy per PC, we use the Equation 4 for merging. The variable o indicates old counter value under input X, while n represents new counter value under input Y. The variable l indicates the number of Prophet loops, where each execution of step 2 counts as one loop, and L is a parameter predefined by the designer.

$$Merged = \begin{cases} o + \frac{1}{\min(l+1,L)} \times (n-o), & \exists o \in X \\ n, & \nexists o \in X \end{cases} \tag{4}$$

For the number of allocated metadata entries at the end of program execution, we apply the Equation 5 for merging:

$$Merged = \max(o, n) \tag{5}$$

We assume a binary first encounters input X (steps 1 and 2), followed by input Y (steps 3 and 2). Next, we will prove that the optimized binary can ultimately adapt to both the previous and the newly observed inputs.

- **Merged prefetching accuracy for Load A**: Since Load A could receive the same hints under input X and Y, both o and n fall within the same range as defined by Equation 1 and Equation 2. After applying Equation 4 with $l = 1$, the merged accuracy remains within the same range, resulting in the same hint being generated in the next step 2.
- **Merged prefetching accuracy for Loads B and C**: Prior to input Y, Prophet lacks counters for Load C (i.e., $\nexists o \in X$). Therefore, the merged prefetching accuracy for Load C is set to n. In the subsequent Step 2, Prophet injects new hints for Load C based on n. As a result, Prophet successfully learns hints for Load C, which was previously unrecorded.
- **Merged prefetching accuracy for Load E**: Load E could receive different hints under input X and Y, causing o and n to fall into different ranges in Equation 1 or Equation 2. In this case, Equation 4 adjusts o through the offset between n and o. If Prophet observes higher prefetching accuracy for Load E under input Y ($n - o > 0$), it increases the estimated accuracy, refining its hints accordingly. Conversely, if $n - o < 0$, Prophet decreases the estimated accuracy. Over time, frequently observed counter values dominate merged results.
- **Merged allocated entries for the entire program**: Prophet adopts a conservative strategy to accommodate the metadata table size requirements for all program inputs.

4.4 Hint Information Injection

According to Equation 1 and Equation 2, each memory instruction requires at most 3-bit hint information. We design two methods for injecting this 3-bit information into memory access instructions.

Hint buffer. Reference to the approach in Whisper [35], we can leverage specialized hint instructions to carry hints. When these hint instructions are executed, Prophet stores the hint information and PC tag in a hint buffer near the temporal prefetcher. Hint instructions are only required to execute once. To minimize their impact on total dynamic instructions, they can be inserted at the entry

[4]we ensure the rounded value does not exceed the maximum number of entries that a 1MB metadata table can accommodate.

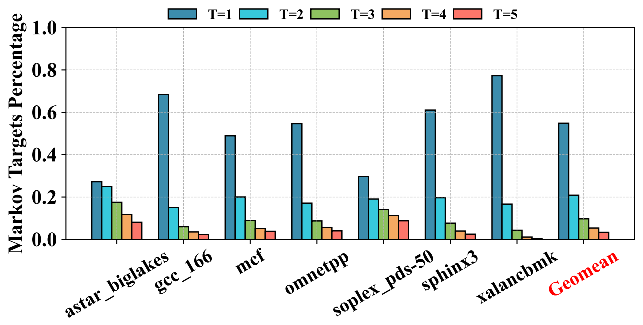


Figure 8: The percentage of Markov target number (T) in temporal prefetching.

point of programs using BOLT [48]. Prophet applies its optimizations to memory instructions whose PC matches an entry in the hint buffer. To efficiently utilize the hint buffer, Prophet focuses on memory instructions that contribute the most to cache misses. We pinpoint these instructions by using `MEM_LOAD_RETIRED.L2_MISS` event at step 1. Empirical findings show that a 128-entry hint buffer (0.19 KB) is sufficient for achieving high performance. Although this approach introduces additional storage overhead, it is compatible with all instruction set architectures.

Reserved bits or instruction prefix. We can embed hints into reserved bits within instructions, allowing hints to be decoded and combined with memory access instructions. This approach saves additional space for storing hints but is constrained by the requirement that commonly used memory access instructions include reserved bits, limiting its applicability. For CISC architectures like x86, we can also add prefixes for memory instructions to carry hints. While this method does not require reserved bits, it increases the code footprint and may impact I-cache performance. However, Prophet focuses on at most 128 memory instructions and introduces only a 3-bit prefix for these instructions. Therefore, Prophet maximally introduces $\frac{3\times128}{64} = 6$ Byte storage overhead to I-cache (usually with 64 KB). Consequently, the x86 instruction prefix scheme has an almost negligible impact on I-cache performance.

4.5 Multi-path Victim Buffer

We observe that the same memory address can appear in multiple distinct temporal patterns. For example, if memory access sequences (A, B, C) and (A, B, D) exhibit temporal patterns, B has two potential Markov targets: C and D. As shown in Figure 8, 54.85%, and 20.88%, 9.71% of memory addresses in the SPEC CPU benchmark have 1, 2, and 3 Markov targets, respectively. However, previous on-chip temporal prefetchers [7, 56, 57] store only one target per Markov entry, often resulting in inaccurate prefetches and unsolvable demand accesses. To address this issue, simply storing multiple prefetching candidates in the metadata table is impractical, as it significantly increases storage overhead.

In order to efficiently handle the above scenario, Prophet introduces a Multi-path Victim Buffer, allowing it to store Markov targets that have been evicted from the metadata table. Prophet manages Multi-path Victim Buffer based on the following rules:

- **Insertion.** To efficiently utilize the space of Multi-path Victim Buffer, we store only Markov targets whose priority levels (Equation 2) are greater than 0 ($acc > EL\_ACC$).

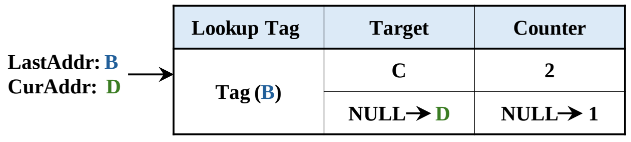


Figure 9: The Multi-path Victim Buffer.

- **Replacement.** We reuse Prophet Replacement Policy to maintain frequently used Markov targets. As shown in Figure 9, we add a counter for each Markov target, incrementing the counter value each time the target is accessed. We set the priority levels as the maximal[5] counter value of Markov targets (which differs from Equation 2).
- **Prefetch.** Prefetching with the Reuse Buffer or metadata table will also trigger a lookup in the Multi-path Victim Buffer. Prophet uses the same lookup addresses to search for entries in the Multi-path Victim Buffer. If Prophet detects different Markov targets, it prefetches these targets accordingly.

5 Evaluation

5.1 Experimental Setup

Table 1: System Configuration.

Module	Configuration
Core	5-wide fetch, 5-wide decode 10-wide issue, 10-wide commit 120-entry IQ, 85/90-entry LQ/SQ 288-entry ROB
Private L1 I/D cache	64 KB each, 4-way, 64B line, 16 MSHRs PLRU, 2 cycles hit latency degree-8 stride prefetcher for L1D cache
Private L2 cache	512 KB, 8-way, 64B line, 32 MSHRs PLRU, mostly_inclusive 9 cycles hit latency
Shared L3 cache	2 MB/core, 16-way, 64B line, 36 MSHRs CHAR [18], mostly_exclusive 20 cycles hit latency
Memory	LPDDR5_5500_1x16_BG_BL32 Single channel, 1 rank per channel

System Configuration. We evaluate Prophet using gem5's FS mode [14]. We utilize facilities within gem5 to collect counters required by Prophet (Section 4.1). Following the rules defined in Section 4.2, we use an offline script to analyze these counters and generate hints, which are then injected into binaries via the hint buffer (Section 4.4). Our simulation environment adopts parameters almost consistent with those utilized in the Triangel [7]. The primary system configurations are outlined in Table 1. Prophet and Triangel are trained on the L2 cache access stream, including prefetch requests generated by L1 stride prefetchers.

Workloads. Following previous temporal prefetchers [7, 56–58] and software indirect prefetching schemes [6, 29, 33, 60], we evaluate Prophet with irregular SPEC CPU workloads and graphic workloads (i.e., CRONO [5]). These workloads exhibit diverse memory access patterns that are representative of a wide range of benchmarks. We apply the SimPoint technique [51] to generate checkpoints across all workloads. Each SimPoint-sampled checkpoint is

[5]If we buffer two or more Markov targets per metadata entries

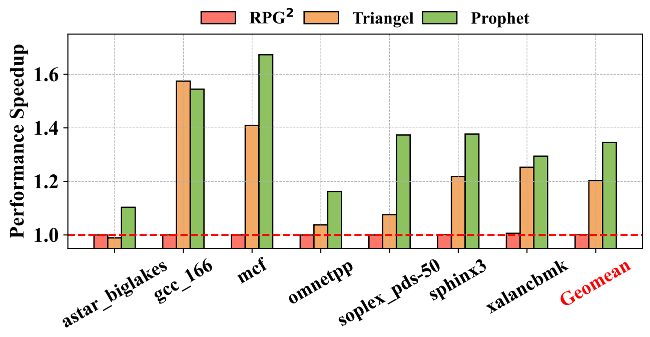


Figure 10: IPC speedup compared to RPG^2 and Triangel.

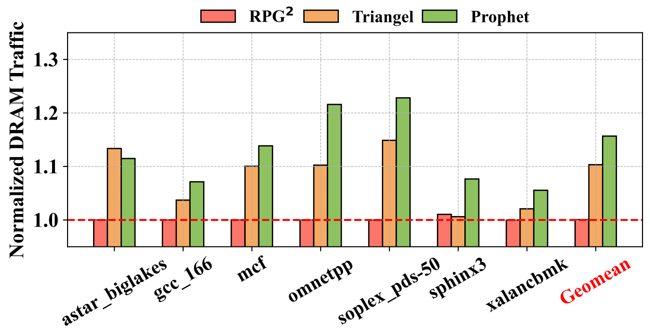


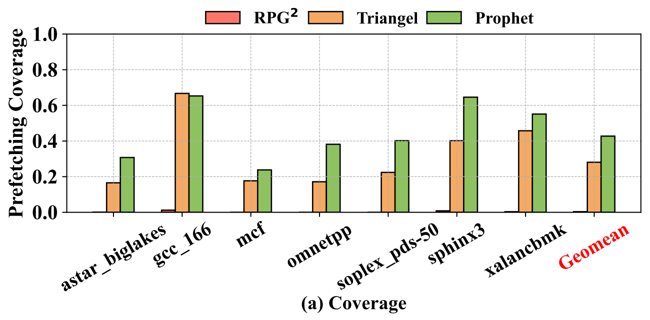
Figure 11: DRAM traffic compared to RPG^2 and Triangel.

warmed up with 250M instructions, followed by a simulation of the next 50M instructions. The reported performance metrics for each benchmark are calculated by aggregating the results from all its checkpoints with weighted averages.

Baseline. We compare Prophet against the state-of-the-art hardware temporal prefetcher, Triangel [7], and software indirect indirect prefetching scheme, RPG^2 [60]. For Triangel, we utilize the open-source implementation provided by its original paper [4], preserving its complete functionality. For RPG^2, we follow its original methodology, first identifying memory instructions that result in at least 10% cache misses and have prefetch kernels supported by RPG^2. Then, we utilize the hint buffer mechanism (Section 4.4) to simulate prefetch instruction insertion. Specifically, we record the PC of identified memory instructions along with an initial prefetch distance in the hint buffer. Upon encountering recorded PCs, we issue a prefetch request where the target address equals the accessed memory address + distance. Finally, we tune the distance using RPG^2's binary search method and record the performance with the optimal distance as the final report.

5.2 Performance

Figure 10 compares the performance of Prophet with RPG^2 and Triangel. The results show that Prophet achieves a 34.58% speedup over the baseline without temporal prefetchers, compared to 0.1% for RPG^2 and 20.35% for Triangel, outperforming RPG^2 by 34.48% and Triangel by 14.23%. Figure 11 shows that Prophet induces 18.67% memory traffic (cumulative DRAM reads + DRAM writes), compared to 0.07% for RPG^2 and 10.33% for Triangel, indicating that Prophet's performance gain over Triangel results in only 5.35%



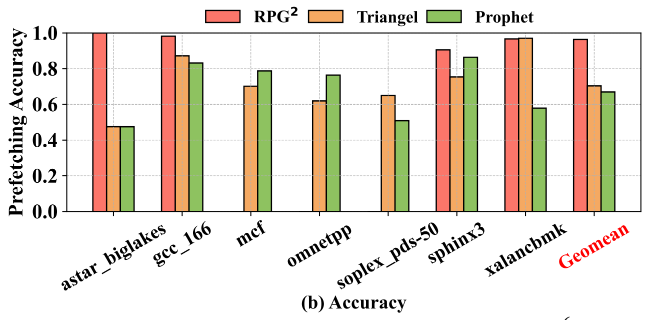


Figure 12: Prefetching coverage and accuracy[6].

additional memory traffic. The workload-specific performance improvements are comparable to those reported in the original Triangel paper [7]. However, the overall speedup for Triangel in our experiments is not identical because we use SimPoint to generate checkpoints instead of the original method described in [4], which evenly samples checkpoints throughout the program's lifecycle, potentially misrepresenting actual program execution.

Analysis: Prophet versus RPG^2. Our experimental results demonstrate that prior profile-guided indirect prefetching schemes are ineffective for most temporal patterns. We observe that most active memory access instructions (i.e., those causing >90% cache misses) in the evaluated workloads exhibit pointer-chasing patterns or indirect access patterns where the prefetch kernel does not follow stride patterns. As described in Section 2.2, prior solutions cannot handle these cases, leading to limited performance improvements.

Analysis: Prophet versus Triangel. Prophet consistently outperforms Triangel across most workloads. To validate this, we analyze the temporal prefetcher's prefetching coverage and accuracy, as shown in Figure 12(a) and Figure 12(b). Prophet reduces demand misses by 42.75%, compared to 28.08% for Triangel. For prefetching accuracy, Prophet performs comparably to Triangel, indicating that Prophet's performance gain comes from more efficient metadata storage management rather than aggressive prefetching. For example, Prophet can simultaneously enhance both prefetching coverage and accuracy in workloads such as mcf and omnetpp. In the case of gcc, which is particularly sensitive to cache pollution, Prophet's performance gain is slightly lower than Triangel.

5.3 Adaptable: Different Program Inputs

Figure 13 evaluates Prophet's adaptability by making it iteratively learn counters from different program inputs within a single application. The leftmost bar shows the performance of Triage4 +

[6]RPG^2 does not identify qualified prefetch kernels for mcf, omnetpp, and soplex, so we set their prefetching accuracy to 0.

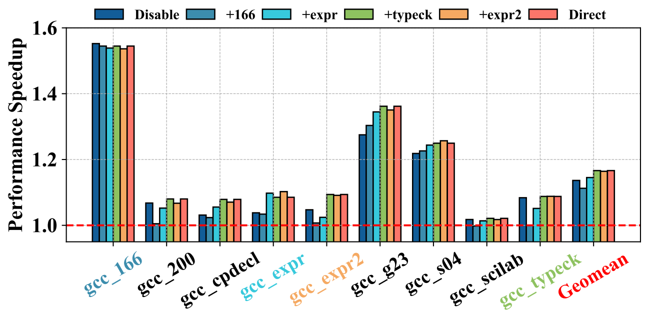


Figure 13: Prophet learns counters from gcc's inputs.

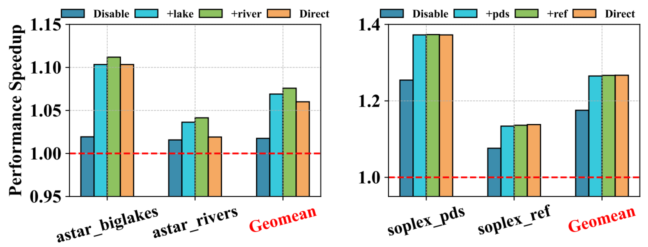


Figure 14: Prophet's learning feature can be generalized to other workloads, such as astar and soplex.

Triangel metadata (Section 5.9), indicating the status where no input is fed to Prophet (shown as "Disable"). Then, we provide the inputs in the following order: gcc_166, gcc_expr, gcc_typeck, and gcc_expr2. The first input, gcc_166, is processed in Step 1, while the subsequent inputs are integrated in Step 3, as shown in Figure 5. To evaluate Prophet's "ideal" performance for each input, we directly profile these inputs and make Prophet learn their respective counters (learning goal, shown in the rightmost bar as "Direct").

Figure 13 demonstrates that, through repeated learning, Prophet enables a single optimized binary to achieve optimal performance across various program inputs. Initially, when learning only from gcc_166, Prophet delivers sub-optimal performance on gcc_expr, gcc_typeck, and gcc_expr2. However, as Prophet incorporates additional inputs, it progressively achieves optimal performance across all of them. Notably, even without directly learning from inputs like gcc_200, Prophet improves its performance due to counters learned from other inputs, such as gcc_expr, which share similar memory access patterns. These results indicate that Prophet can achieve optimal performance across all program inputs with fewer training iterations than the total number of inputs. Consequently, with only 4 rounds of learning, Prophet achieves near-optimal performance across all 9 gcc inputs. Furthermore, Figure 14 demonstrates the learning features in Prophet can be generalized to other workloads.

Prophet versus Other profile-guided solutions. To the best of our knowledge, existing profile-guided solutions lack an adaptive mechanism for learning counters or traces from different inputs. These solutions either deliver sub-optimal performance when encountering new inputs or require fresh profiling (e.g., RPG^2) but cannot leverage prior profiling data, making them unable to adapt to previously encountered inputs. In contrast, Prophet's adaptability makes it more practical for deployment in commercial processors.

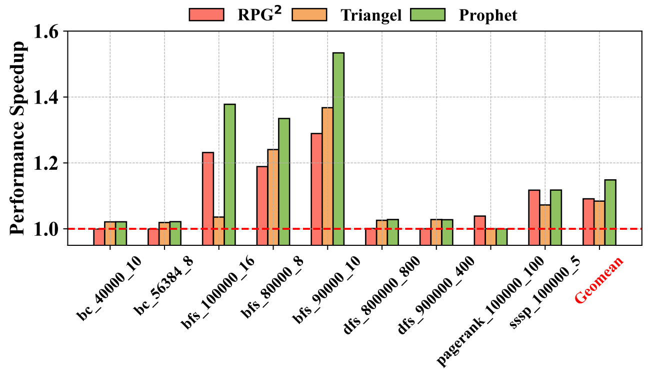


Figure 15: IPC speedup on graph workloads.

5.4 Lightweight: Profiling, Analysis, Instruction Overhead

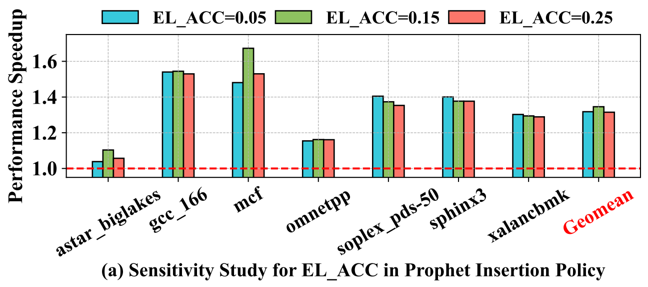
5.4.1 Profiling Overhead. Prophet's profiling overhead arises from collecting counters during Step 1 and Step 3 in Figure 5. Prophet utilizes the PEBS and standard PMU to gather these counters (Section 4.1), so the profiling overhead depends on the implementation of PEBS and PMU. According to [15], sampling 4 PEBS events incurs less than 2% performance overhead, while sampling a single standard PMU event incurs negligible overhead. Given Prophet only requires sampling two or three PEBS events (depending on the ways to inject hints) and one standard PMU event, it incurs less than 2% profiling overhead. Importantly, not every program execution requires profiling. Prophet only sample counters at intervals (Section 3.2), determined by the complexity of programs and input variety. Empirically, profiling once every 10–100 executions suffices. Most program executions incur no profiling overhead. Moreover, we can stop profiling when further performance gains are minimal.

5.4.2 Analysis Overhead. Prophet's analysis overhead comes from analyzing the collected counters to generate hints during Step 2 in Figure 5. Across all evaluated workloads, Prophet's analysis overhead is negligible, less than one second. Furthermore, as with profiling, not every program execution incurs analysis overhead; only those executions in which Prophet is enabled to collect counters will require analysis, further reducing the overall analysis overhead.

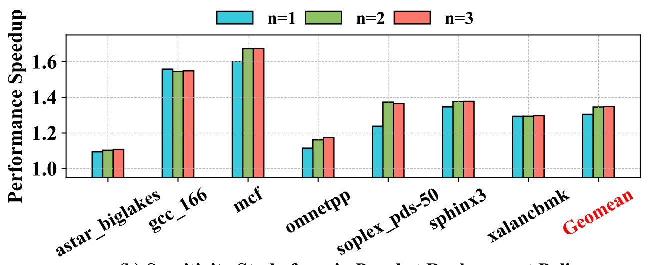
5.4.3 Instruction Overhead. Prophet's instruction overhead depends on the hint injection methods described in Section 4.4. If reserved bits within instructions are used to carry hints, Prophet incurs no additional instruction overhead. In contrast, if specialized hint instructions are employed, the instruction overhead corresponds to the number of inserted hint instructions. In our setup, we insert a maximum of 128 hint instructions at the entry point of programs (Section 4.4). Compared to evaluated SPEC CPU workloads containing billions of instructions [44], Prophet introduces almost negligible overhead for overall static and dynamic instructions.

5.5 Generalization: Graphic Workloads

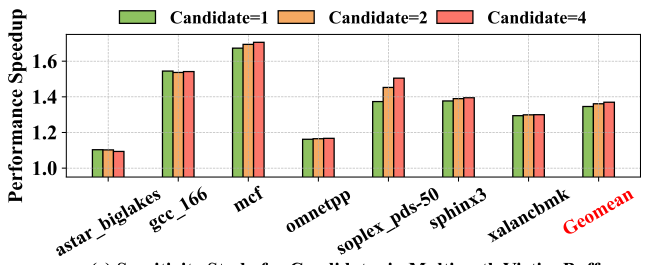
Figure 15 evaluates Prophet's performance using CRONO [5], a benchmark suite widely employed in indirect access prefetching schemes [6, 29, 33, 60]. Experimental results demonstrate that Prophet provides a performance speedup of 14.85% over the baseline with a hardware stride prefetcher alone, compared to 9.11% for RPG^2



(a) Sensitivity Study for EL_ACC in Prophet Insertion Policy



(b) Sensitivity Study for n in Prophet Replacement Policy



(c) Sensitivity Study for Candidates in Multi-path Victim Buffer

Figure 16: Sensitivity study.

and 8.41% for Triangel. Unlike SPEC CPU benchmarks, CRONO features more prefetch kernels with stride patterns, aligning with RPG^2's strengths. As a result, RPG^2 delivers greater performance gains on CRONO. Prophet outperforms RPG^2 by handling more complex temporal patterns beyond RPG^2's scope.

5.6 Sensitivity: Parameters in Prophet

Figure 16 evaluates primary parameters[7] in Prophet, such as *EL_ACC* for the Prophet Insertion Policy, *n* for the Prophet Replacement Policy, and the candidates per metadata entry for the Multi-path Victim Buffer.

For *EL_ACC*, we observe that both too high and too low parameter values can negatively impact performance. A low *EL_ACC* causes the metadata table to buffer too many metadata entries that do not exhibit temporal patterns, while a high *EL_ACC* has the opposite effect, potentially filtering out valuable entries.

For *n*, we observe that introducing additional bits to enable fine-grained classification of temporal patterns results in improved performance. However, the performance improvement is limited, and this modification introduces additional storage overhead for the Prophet Replacement State. To balance the performance gain with the storage overhead, we adopt a moderate configuration (n = 2, 2-bit Prophet Replacement State).

For the number of Markov candidates in the Multi-path Victim Buffer, we observe that having one candidate per metadata entry achieves the best trade-off between performance gain and storage overhead. While prefetching more candidates improves prefetching coverage, it can negatively impact performance. For example, prefetching additional Markov targets causes performance slowdown in astar, which is sensitive to cache pollution and memory bandwidth wastage. Furthermore, as shown in Figure 8, a large proportion of memory addresses tend to have fewer than two Markov targets, resulting in unnecessary storage overhead when excessive Markov candidates are reserved in the buffer.

[7]The green bar represents the parameters used in other experiments.

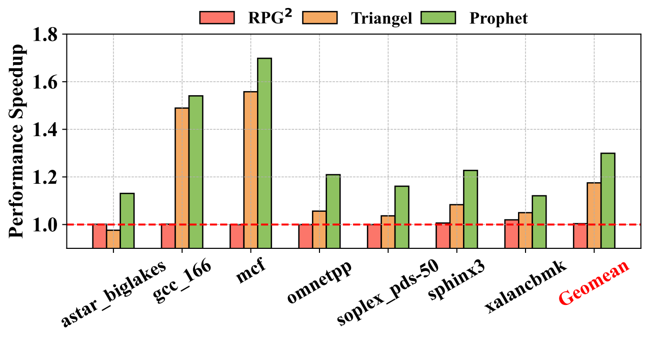


Figure 17: IPC speedup with varying L1 prefetcher.

5.7 Sensitivity: Impact of L1 Prefetcher

Figure 17 evaluates Prophet's performance with varying L1 prefetchers. As outlined in Table 1, our system configuration aligns with Triangel's setup for consistent comparison. However, commercial processors typically include additional L1 prefetchers since they can leverage more information (e.g., virtual addresses) and prefetch across page boundaries. For example, Arm's Neoverse V2 [2] integrates stream, stride, and spatial prefetchers in the L1 cache. To assess Prophet's performance with a more realistic L1 prefetcher configuration, we replace the L1 stride prefetcher with IPCP [47], simulating the setup in Arm's Neoverse V2.

Figure 17 demonstrates that Prophet outperforms both RPG^2 and Triangel, achieving a performance speedup of 29.95% over the baseline without a temporal prefetcher. In comparison, RPG^2 shows a modest speedup of 0.36%, while Triangel achieves a speedup of 17.51%. Prophet consistently outperforms other schemes across all evaluated workloads. These results demonstrate that Prophet's performance improvement can also be applied to more complex L1 prefetcher configurations.

5.8 Sensitivity: Memory Bandwidth

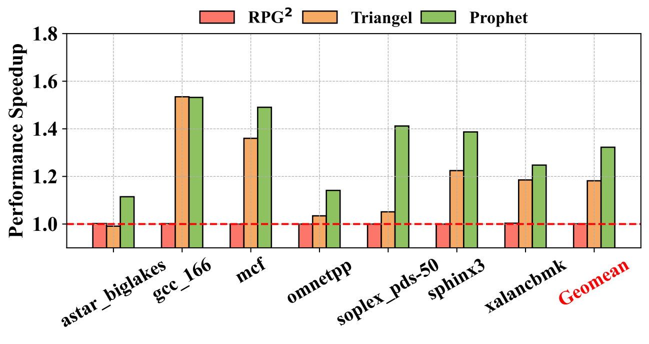
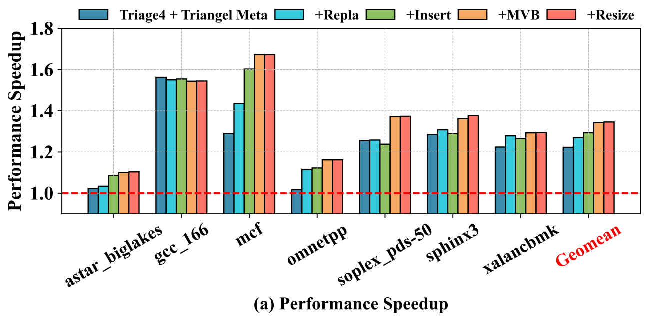


Figure 18: IPC speedup with varying DRAM channels.



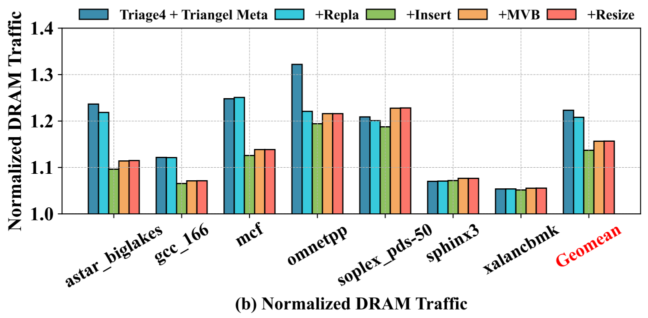


Figure 19: Prophet Features Breakdown.

Figure 18 evaluates Prophet, RPG^2, and Triangel with an increased number of DRAM channels. Prophet provides a performance speedup of 32.27% over the baseline without temporal prefetchers, compared to 0.1% for RPG^2 and 18.17% for Triangel. The experimental results show that Prophet remains effective across varying memory bandwidth environments.

5.9 Prophet Features Breakdown

Figure 19 illustrates the contribution of each feature provided by Prophet. Our ablation study begins with Triage at a prefetch degree of 4 [57], combined with Triangel's metadata format. Overall, Prophet's replacement policy, insertion policy, and Multi-path Victim Buffer contribute the most to performance speedup. Meanwhile, Prophet resizing proves most effective for workloads with relatively small metadata table requirements, such as sphinx3.

Prophet replacement policy is effective across most workloads by providing fine-grained management: it prioritizes the retention of metadata entries that achieve higher prefetching accuracy. Workloads with large working sets that are sensitive to cache pollution, such as mcf and omnetpp, particularly benefit from this feature, achieving performance gains of 14.53% and 9.89%, respectively. Additionally, the Prophet replacement policy also proves beneficial in reducing memory traffic for workloads like astar and omnetpp.

Prophet insertion policy is conservatively designed to avoid filtering out useful metadata (Section 4.2). Unlike the *PatternConf* mechanism in Triangel, Prophet insertion policy avoids significant performance drops for omnetpp, soplex, and sphinx3. Meanwhile, it yields a significant performance speedup for mcf (16.72%) and effectively reduces memory traffic across various workloads.

Multi-path Victim Buffer is developed to predict complex temporal patterns where a single memory address has multiple potential Markov targets, providing an opportunity to reduce more demand misses compared to the original metadata format design. This feature contributes to performance improvements across multiple workloads, with soplex showing a notable 13.46% performance speedup. The Multi-path Victim Buffer only slightly increases memory bandwidth by 1.95% due to our fine-grained management. Consequently, the Multi-path Victim Buffer enhances the performance of astar (which is sensitive to constrained memory bandwidth) rather than negatively impacting it.

Prophet Resizing allocates metadata table space based on application requirements, allowing more space for the LLC without compromising metadata table capabilities. This approach yields performance benefits, especially for applications with smaller metadata needs. For example, sphinx3, which requires less than 1 MB of metadata table, achieves a 1.5% performance gain.

The flexibility of Prophet. Prophet's features are designed to be modular, allowing programmers to selectively enable or disable specific features based on evaluated performance and memory traffic. This adaptability means that if Prophet's impact on performance is unfavorable for certain workloads, programmers can selectively roll back to a subset of Prophet's features or revert to the runtime temporal prefetcher, such as gcc_166.

5.10 Storage Overhead

The storage overhead of Prophet stems from its replacement state, the hint buffer, and the Multi-path Victim Buffer.

- **Prophet replacement stats.** Prophet supports a maximum metadata table of 1 MB, or 196,608 entries, with each entry requiring a 2-bit replacement state. Consequently, the total storage overhead for Prophet's replacement states is 48 KB.
- **Hint buffer.** If a hint buffer is used for processing hint information (Section 4.4), Prophet requires extra storage for this buffer. Our experiments indicate that a 128-entry hint buffer is sufficient for high performance, adding 0.19 KB.
- **Multi-path Victim Buffer** requires an additional 43 bits of storage per metadata entry: 31 bits for the memory address, 10 bits for the tag, and 2 bits for Prophet's replacement policy counter. For a buffer with 65,536 entries, this results in a storage overhead of 344 KB. We compare the performance gain of allocating this additional storage to the LLC, observing that the Multi-path Victim Buffer achieves an extra 2.21% performance improvement (4.95% vs. 2.74%).

5.11 Energy Overhead

We evaluate the energy overhead of Prophet with a focus on the memory hierarchy. We utilize CACTI [43] to model the energy consumption of the on-chip memory hierarchy under a 22 nm technology node, and estimate the DRAM access energy overhead to be 25× that of the LLC access overhead, similar to Triangel [7]. Our experiments show that Prophet only introduces 1.6% energy overhead for the memory hierarchy compared to Triangel. Given that Prophet's performance improvement over Triangel is 14.23%, the 1.6% energy overhead is relatively negligible.

6 Related Works

In this section, we discuss the most relevant work in hardware temporal prefetching and profile-guided prefetching.

Hardware temporal prefetchers. The most related work to Prophet is hardware temporal prefetchers [7, 10, 26, 46, 55–58]. Like these prefetchers, Prophet requires metadata storage to maintain correlations between memory addresses. However, to the best of our knowledge, Prophet is the only approach that integrates profile-guided techniques into metadata storage management. By leveraging future knowledge, Prophet significantly optimizes metadata management while avoiding expensive hardware modifications. Furthermore, Prophet is fully compatible with existing hardware temporal prefetchers, allowing chip designers to flexibly choose between Prophet and hardware prefetchers (Section 5.9).

Profile-guided prefetching. Profile-guided techniques have been applied to various aspects of prefetching, such as software prefetching [8, 29, 33, 60], criticality-aware prefetching [40], and instruction prefetching [34]. Profile-guided software prefetching schemes identify prefetch kernels and utilize software prefetch instructions to predict them. Profile-guided criticality-aware prefetching focuses on identifying demand misses that lead to ROB stalls and extends the instruction scheduler to prioritize critical instructions. Profile-guided instruction prefetching schemes precisely identify I-cache misses and combine multiple non-contiguous prefetches into a single prefetch instruction.

Prophet sets it apart from these existing profile-guided prefetching schemes by focusing on a distinct aspect of prefetching: temporal prefetching, an area where prior techniques fall short. Additionally, Prophet introduces innovative methodologies for adaptable, lightweight, and compatible profile-guided optimizations, making it more seamless and practical to implement in industrial applications.

7 Conclusion

In this paper, we propose Prophet, an adaptable, lightweight, and compatible profile-guided solution for temporal prefetching. Prophet injects hints into programs to guide the temporal prefetcher's replacement policy, insertion policy, and resizing operations, while dynamically tuning these hints to allow a single optimized binary to adapt to various program inputs. Our evaluations demonstrate that Prophet outperforms the state-of-the-art hardware temporal prefetcher and software indirect memory access prefetching scheme, with negligible profiling, analysis, and instruction overheads. Prophet demonstrates superior performance across a wide range of benchmarks and configurations.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments and suggestions. This research was supported by National Natural Science Foundation of China (NSFC) 62304192, Hong Kong Research Grants Council (RGC) YCRG Grant C6003-24Y, and ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government. We thank HKUST Fok Ying Tung Research Institute and National Supercomputing Center in Guangzhou Nansha Sub-center for computational resources.

References

[1] [n. d.]. Intel's PerfMon Events. https://perfmon-events.intel.com.

[2] 2023. Hot Chips 2023: arm's neoverse v2. https://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC2023.Arm.MagnusBruce.v04.FINAL.pdf.

[3] 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[4] 2024. Github: gem5-triangel. https://github.com/SamAinsworth/gem5-triangel.

[5] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*. 44–55.

[6] Sam Ainsworth and Timothy M Jones. 2017. Software prefetching for indirect memory accesses. In *CGO*. 305–317.

[7] Sam Ainsworth and Lev Mukhanov. 2024. Triangel: A High-Performance, Accurate, Timely On-Chip Temporal Prefetcher. In *ISCA*.

[8] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *ASPLOS*. 513–526.

[9] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 176–186.

[10] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *HPCA*. 131–142.

[11] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *HPCA*. 399–411.

[12] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Evaluation of hardware data prefetchers on server processors. *CSUR* (2019), 1–29.

[13] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. Dspatch: Dual spatial pattern prefetcher. In *MICRO*. 531–544.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* (2011).

[15] Georgios Bitzes and Andrzej Nowak. 2014. The overhead of profiling using PMU hardware counters. *CERN openlab report* (2014), 1–16.

[16] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* (1970), 422–426.

[17] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 40–52.

[18] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *PACT*. 293–304.

[19] William Y Chen, Scott A Mahlke, Pohua P Chang, and Wen-mei W Hwu. 1991. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *MICRO*. 69–73.

[20] Fredrik Dahlgren and Per Stenstrom. 1995. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *HPCA*. 68–77.

[21] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*. 1–42.

[22] Edward H Gornish, Elana D Granston, and Alexander V Veidenbaum. 1990. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ICS*. 128–142.

[23] Mingjian He, Hua Wang, Ke Zhou, Kaichao Cui, Huabing Yan, Chang Guo, and Rongfeng He. 2022. DSDP: Dual Stream Data Prefetcher. In *PACT*. 372–383.

[24] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *MICRO*. 397–408.

[25] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In *ICS*. 499–500.

[26] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*. 247–259.

[27] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News* (2016), 78–89.

[28] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH computer architecture news* 38, 3 (2010), 60–71.

[29] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Eurosys*. 747–764.

[30] Shizhi Jiang, Qiusong Yang, and Yiwei Ci. 2022. Merging similar patterns for hardware prefetching. In *MICRO*. 1012–1026.

[31] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[32] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam,

Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-guided btb prefetching for data center applications. In *MICRO*. 816–829.

[33] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *OSDI*. 163–181.

[34] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-spy: Context-driven conditional instruction prefetching with coalescing. In *MICRO*. IEEE, 146–159.

[35] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. 2022. Whisper: Profile-guided branch misprediction elimination for data center applications. In *MCIRO*. 19–34.

[36] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided instruction cache replacement for data center applications. In *ISCA*. 734–747.

[37] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *MICRO*. 1–12.

[38] Sunil Kim and Alexander V Veidenbaum. 1997. Stride-directed prefetching for secondary caches. In *ICPP*. 314–321.

[39] Mengming Li, Qijun Zhang, Yongqing Ren, and Zhiyao Xie. 2025. Integrating Prefetcher Selection with Dynamic Request Allocation Improves Prefetching Efficiency. In *HPCA*.

[40] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: critical slice prefetching. In *ASPLOS*. 300–313.

[41] Pierre Michaud. 2016. Best-offset hardware prefetching. In *HPCA*. 469–480.

[42] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *CSUR* (2016), 1–35.

[43] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[44] Arun A Nair and Lizy K John. 2008. Simulation points for SPEC CPU 2006. In *ICCD*. 397–403.

[45] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an accurate local-delta data prefetcher. In *MICRO*. 975–991.

[46] Kyle J Nesbit and James E Smith. 2004. Data cache prefetching using a global history buffer. In *HPCA*. 96–96.

[47] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *ISCA*.

[48] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *CGO*. 2–14.

[49] Biswabandan Panda. 2023. CLIP: Load Criticality based Data Prefetching for Bandwidth-constrained Many-core Systems. In *MICRO*. 714–727.

[50] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *HPCA*. 626–637.

[51] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices* (2002), 45–57.

[52] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *MICRO*. 141–152.

[53] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. *ACM SIGARCH Computer Architecture News* (2006), 252–263.

[54] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *ISCA*. 742–756.

[55] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *HPCA*. 79–90.

[56] Hao Wu, Krishnendra Nathella, Matthew Pabst, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2021. Practical temporal prefetching with compressed on-chip metadata. *IEEE Trans. Comput.* (2021), 2858–2871.

[57] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *MICRO*. 996–1008.

[58] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *ISCA*. 449–461.

[59] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[60] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. RPG2: Robust Profile-Guided Runtime Prefetch Generation. In *ASPLOS*. 999–1013.